A SOFTWARE ENGINEERING VIEW OF THE FLIGHT DYNAMICS ANALYSIS SYSTEM (FDAS)

Prepared for

GODDARD SPACE FLIGHT CENTER

By

D. Card, W. Agresti, L. Jordan and V. Church

COMPUTER SCIENCES CORPORATION

Under

Contract NAS 5-27888
Task Assignment 40500

DECEMBER 1983

TABLE OF CONTENTS

SECTION 1 - INTRODUCTION

The Flight Dynamics Analysis System (FDAS) is intended to provide an integrated software development support environment for research applications in the areas of orbit, attitude, and mission analysis. FDAS was conceived to assist users in the preparation, execution, and interpretation of software experiments. This memorandum provides an assessment of FDAS at one step in the requirements definition process--a prototype support environment.

1.1 The FDAS "SUPPORT ENVIRONMENT" CONCEPT

Several independent systems now provide parts of the flight dynamics experiment capabilities that FDAS is to encompass. These systems include the Research and Development Goddard Trajectory Determination System (R&D GTDS), the Goddard Mission Analysis System (GMAS), the Research and Development Mission Analysis System (RADMAS) and the Attitude Dynamics Generator (ADGEN). Some of the software comprising these systems will be adapted for FDAS.

These existing systems are awkward to use for experimentation because of the difficulty in modifying the software for a particular study. A detailed knowledge of both the research software and the computer system on which it runs is required for typical investigations, even when the modification involved is relatively clear and compartmentalized. FDAS is to provide a user-friendly executive for selecting and combining elements from a common software library to perform any required experiment. FDAS is intended to supplant the R&D GTDS, GMAS, RADMAS, and ADGEN systems.

1.2 THE FDAS PROTOTYPE EFFORT

The "support environment" concept of FDAS is unlike that of any system previously developed in the GSFC Flight Dynamics area. As a consequence, the requirements for the system were

unclear and the magnitude of the needed software development effort unknown. To clarify these issues, a prototype of the proposed FDAS system was developed. The prototyping effort was part of the requirements definition process, where issues and concepts could be tried and tested without committing the full system to potentially flawed approaches. Development of the prototype was expected to aid in defining the scope of the full FDAS system, both in functionality and magnitude.

1.3 EVALUATION EFFORTS

This memorandum describes one of four evaluations of the FDAS concept and prototype. Each evaluation team has approached FDAS from a different perspective. Academicians are assessing it as a case study of the prototyping approach to software; potential users are judging its utility and flexibility; the developers are evaluating the quality of the implementation and design; and this group of reviewers is attempting to take a software engineering point of view.

In discussions, in literature research, in exercising the prototype system, we tried to address FDAS at four levels: the "support environment" concept, the specific approach implied by the FDAS prototype, the prototype as a requirements definition tool, and the usability of the features provided in the prototype. The specific questions we addressed in our study include:

What are the goals of FDAS, and what criteria can be used in assessing their achievement?

What alternative approaches might be taken to solving the problem that FDAS is intended to address?

How well does the approach taken (that implied by the decisions made in building the prototype) compare with other possibilities?

How effective is the prototype FDAS in meeting its requirements (providing information about the requirements and feasibility of the full FDAS)?

How well do the design concepts employed in the prototype support the end goals of FDAS?

It is our hope that the results of this and other studies of FDAS will aid in the specifications and development planning for the full FDAS system.

1.4 SUMMARY AND OVERVIEW

Section 1 of this assessment report provides the context for evaluation, describing the FDAS concept, the current stage of requirements definition, and our particular orientation. In Section 2 we present the criteria which we defined for use in (relatively) objectifying the FDAS goals of combining functionality and power with ease of use. To provide a basis of comparison, we identified several alternative approaches to satisfying those criteria. Section 3 defines these alternatives and discusses the strengths and weaknesses of each. Based on these comparisons, and on precepts and principles of human engineering, we evaluated the FDAS prototype. Our findings, both on concepts and on their realization, are examined in Section 4. Section 5 presents the conclusions of this evaluation group (summarized below) and its recommendations for future FDAS development.

Our major conclusions can be summarized as follows:

The FDAS concept is an attractive and feasible direction for improving the Flight Dynamics analysis working environment. Similar efforts in other fields have demonstrated their value; flight dynamics need not be an exception.

The "software builder" approach taken in the FDAS prototype is a feasible solution to the original problem; but it is not the only solution, not is it clearly superior to other possibilities.

The FDAS prototype effort is a poor example of a prototype; rather than painting in broad brush strokes so that competing concepts and approaches can be evaluated, the FDAS prototype presents one excessively fine view of a potential solution. Where a prototype should serve as a tool or test bed for requirements defintion, the FDAS effort almost requires a "yes/no" assessment.

The user interface provided by the FDAS prototype is exceedingly difficult to use; repeated attempts to make productive use of the system, with extensive reference to the user's manual but without recourse to the developers, were unsuccessful. The user interface should be the central element of an interactive prototype; in this case it was an obstacle to evaluation of FDAS.

SECTION 2 - ASSESSMENT CRITERIA

Because our purpose is to assess the basic concept underlying FDAS (as well as the FDAS prototype), we began by restating the initial problem in terms of assessment criteria. These criteria provide an independent (and corroborative) view of the goals and intended purpose of the system; they represent a less "implementation-directed" view of the general FDAS requirements than is presented in the prototype documentation. These criteria provide the basis for our evaluation of the FDAS prototype, the FDAS approach as embodied in the prototype, and alternative approaches to addressing the original problem.

We considered that the initial driving requirement is for a tool or support environment which would let an analyst make and test modifications to (largely) preexisting models without having to leave the level of abstraction required of the analysis. That is, a comparative study of orbit propagation algorithms should not have to deal with the level of files and compilers and "housekeeping" code - it should deal with orbit algorithms and the mechanizations thereof. The assessment criteria described below reflect our understanding of how this requirement might be satisfied.

Seven criteria were identified; these are listed here and elaborated below:

- Requirement (of the user) for global knowledge of the application software.
- Requirement for new ancillary learning to use the system.
- Flexibility and accessibility of provided capabilities.
- Ease of application-level modifications.
- Effort required for system-level modifications.

- Data/software/analysis integration of the support environment.
- Feasibility and cost of implementation.

2.1 REQUIREMENT FOR GLOBAL KNOWLEDGE OF APPLICATION SOFTWARE

A significant problem with the current approach to testing new conditions is the lack of separability of the software involved. A user should be required only to understand how his or her particular concern relates to the rest of the system in an algorithmic sense (at the application level of abstraction), without learning the details of how data is transferred and COMMONs assigned. This criterion measures how much of the mechanics of the application system must be understood in order to make use of the operative elements of the system. Minimizing this measure is desirable as it allows the analyst to operate at the problem level instead of the process level.

2.2 REQUIREMENT FOR NEW SYSTEM KNOWLEDGE

In order to use the present applications systems, the user needs to know how to edit, compile, and execute programs on the host computer(s); this is typically not different from other computer related tasks, and does not require any conceptually new knowledge of the support system. A new, comprehensive support environment may require that users learn new skills and terminology to avail themselves of the benefits and capabilities. The FDAS should minimize this new learning requirement so as not to pose a hurdle to its initial use.

2.3 APPLICATION-LEVEL FLEXIBILITY AND ACCESSIBILITY

A primary purpose of FDAS is to support modifications to previously developed models, simulators, experiments, and algorithms. This criterion measures the degree to which the common library of software is accessible for change and

experimentation. Low-level functionality is the primary concern here; details of input and output or the precision of calculations may be subject to experimentation. While major components (such as a drag model) are expected to be replaced, utility functions may not be as readily accessible.

2.4 EASE OF APPLICATION-LEVEL MODIFICATIONS

The nature of the analytical efforts to be supported requires that data or software or both be modified and tested. A critical requirement for FDAS is to facilitate such modifications. Typical changes include supplying a new set of initial conditions, or changing the convergence control on an interation, or replacing a major component (such as a drag model).

2.5 EFFORT FOR SYSTEM-LEVEL MODIFICATIONS

Evolution of the working environment may lead to requirements for new supporting or utility functions, new analysis tools, different I/O capabilities, new system-level functionality. This criterion addresses the difficulty of adding such new features to the support system. (It is assumed that such additions would typically be performed by the implementation or maintenance group - not by users).

2.6 DATA/SOFTWARE/ANALYSIS INTEGRATION

As a support environment, FDAS should assist the user in managing the tests and trials and changes which constitute the analysis activity. FDAS should provide automatic facilities for logging results, recording initial conditions and software versions, and supporting analytical comparisons. This function is typically performed by the user with the host computer file system and careful notes; this criterion measures how much of that error-prone activity is assumed by the support environment.

2.7 FEASIBILITY AND COST

Although this study group was not directed to evaluate cost *per se*, the relative magnitude of various approaches is an inescapable point of comparison. Studies of similar systems in academic or industrial environments can demonstrate the basic feasibility of such advanced support tools. The complexity and probable effort required for various alternatives is addressed with this criterion.

SECTION 3 - ALTERNATIVES TO FDAS

This section discusses alternative approaches to meeting the FDAS objectives described in the last section. Three major alternatives are developed. There are three variations of each alternative depending on the type of implementation language. The alternatives are rated using the assessment criteria of Section 2.

3.1 DEFINING ALTERNATIVES

The assessment team addressed the questions of postulating candidate software systems to meet the goals of FDAS. Instead of presenting all the early proposals, a list is given below of key questions which illustrate the features that distinquish the alternatives from one another:

- What does the system do for the user?
- How does the system perform its fuctions?
- How does the user interact with the system?
- Does the system provide its own storage management facilities for data sets, programs, or program parts?
- Does the system produce only output data from program execution, or does it generate application program?
- Can any developed programs be extracted from the system and executed independently?
- What is the scope of the system? For example, does it provide facilities for editing and program composition?
- Does the user write code? Or is processing described by user responses to prompts of the system?
- How does the system appear to the user - in some graphical form, as menus, as command language?

These questions were used to identifiy the distinguishing characteristics of the various proposals. Three alternatives emerged as significantly different approaches. Each alternative actually represents a class of systems, differentiated

as to distinguishing features would be elaborated to form operational products.

3.1.1 REDEVELOP EXISTING SOFTWARE

The first alternative is to redevelop existing software to make it more modular and more usable. The software would consist entirely of application programs. No system-executive functions would be implemented. The users would continue to rely on existing systems software (operating system, linker, etc.). We emphasize that this is not a "do nothing" alternative. The approach described here is to repackage existing software functions so that analysts can work with them more easily. New code would be written, especially to improve the user interface. It is expected that some of the technical sections of code would be directly reusable.

The present software - R&D GTDS, GMAS, RADMAS, and ADGEN - would be considered to comprise a single collection of functions; decisions would be made to organize these functions in a more useful way.

The particular language chosen for use in the redevelopment is an important factor, because (in contrast to other alternatives) both developers and users would be working in that language. (In fact, implementation language emerged as an orthogonal concept in the consideration of alternatives. For each of the three alternatives, the language must be addressed.)

FORTRAN is an obvious candidate because it is used currently. A second option is a different existing language, e.g., Pascal or Ada. The assessment team did not consider the availability of compilers, only that using a different language is a reasonable choice. A third language possibility is a special-purpose flight-dynamics language, designed to meet the needs of analysts and other users in the application area.

A special-purpose language would contain data types and operations tailored to flight dynamics. Special-purpose languages are widely used as shown in rosters of programming languages (Reference 1).

3.1.2 COMPREHENSIVE DATA-DRIVE PROGRAM

The second alternative is to develop a comprehensive multi-function program whose behavior is controlled by user option-lists or responses to prompts. The program appears to the user as a self-contained entity - a collection of model types, analytical procedures, and executive support functions. The user doesn't write procedural code to add major new capabilities; instead, the features of the program are implemented so that as many parameters and variations as possible are exposed to the user. The program leads the user through model definition and experiment execution by presenting the user with opportunities to make choices, enter values, or insert one-line functions. The program may be knowledge-based (Reference 2), containing inference rules and questioning the user for the information it needs to arrive at a complete specification of an execution environment.

Some examples from other application areas may help to explain this alternative, although no example should be expected to match perfectly with the FDAS situation. Consider the program used by automobile designers to relate vehicle design to expected miles-per-gallon. The program has a significant graphical component, leading the user through the process of drawing or specifying the external shape of the vehicle. Next the user makes decisions about the composition of materials and the distribution of weight over the vehicle. An analytical model, embedded in the program, outputs the average miles-per-gallon based on vehicle surfaces, weight, and other factors.

As another example consider a single program to simulate the operation of an airline. Many submodels must be included to represent finance, marketing, passenger loading, costs, revenues, routes, etc. The user is lead through the program to answer "what if" questions by responding to system prompts. Other examples of complex, multi-function programs are described in References 3 and 4.

3.1.3 SOFTWARE-BUILDING APPROACH

The third alternative is a software builder system that enables the user to combine new or old software components with data to make complete models. References 5 through 8 contain various features of software builder systems. Because the current FDAS prototype is representative of this class, the alternative will not be discussed here in detail.

3.1.4 SUMMARY

In summary, three approaches have been developed:

- Redevelop existing software
- Comprehensive data-driven program
- Software builder

Furthermore, each approach can be developed under each of three language options:

- FORTRAN
- Another existing language
- Special-purpose language

The three alternatives and three language variations define nine different possibilities for comparison.

3.2 RATING THE ALTERNATIVES

The alternatives of Section 3.1 were evaluated using the criteria developed in Section 2. There were three implementation approaches and each one could use three languages, for a total of nine alternatives.

To obtain a quantitative assessment, the assessment team considered each criterion in turn, ranking the alternatives from one (best) to nine (worst) according to how well the alternative met the criterion. Adjustments were made for ties. Table 3-1 shows the results of the rankings, which should be interpreted in light of the following:

- Although they have been expressed quantitatively, the rankings are, of course, subjective.
- Ordinal measures (rankings) were as definitive as the team wanted to be regarding the comparison of alternatives.
- The alternatives had unequal levels of specification. With the software builder, there exists a particular instance of that approach, viz., the FDAS prototype, where the other two alternatives were understood at the level of Section 3.1.

Before discussing the implications of the results, the remainder of this subsection describes some of the reasoning which led to the rankings in Table 3-1. Each criterion (column in Table 3-1) will be discussed in turn. The relationships that exist in the rankings will be expressed in the following notation:

- Approaches
  - RWS - Redevelop existing software
  - CP - Comprehensive data-driven program
  - SB - Software builder
- Languages (as subscripts)
  - F - FORTRAN
  - O - Other exsting language
  - S - Special-purpose language

ALTERNATIVES	CRITERIA 1 KNOWLEDGE OF APPLICATION	2 LEARNING NEW SUPPORT SYS.	3 FLEXIBILITY	4 EASE OF MODIFYING	5 EASE OF EXTENDING	6 LEVEL OF INTEGRATION	7 EASE OF IMPLEMENTING	TOTAL
– REDEVELOP EXISTING SOFTWARE								
• FORTRAN	9	1	2.5	9	6	8	1	36.5
• OTHER EXIST. LANG.	8	6	2.5	8	5	8	4	41.5
• SPECIAL-PURP. LANG.	7	5	5.5	7	4	8	7	43.5
– COMPREHENSIVE DATA-DRIVEN PROGRAM								
• FORTRAN	2	3	8	2	9	2	2	28.0
• OTHER EXIST. LANG.	2	3	8	2	8	2	5	30.0
• SPECIAL-PURP. LANG.	2	3	8	2	7	2	8	32.0
– SOFTWARE BUILDER								
• FORTRAN	6	7	2.5	6	3	5	3	32.5
• OTHER EXIST. LANG.	5	9	2.5	5	2	5	6	34.5
• SPECIAL-PURP. LANG.	4	8	5.5	4	1	5	9	36.5

TABLE 3-1 RANKING OF ALTERNATIVES AGAINST CRITERIA [a]

[a] 1 = BEST ALTERNATIVE TO SATISFY CRITERION
9 = WORST ALTERNATIVE TO SATISFY CRITERION

For example, the relation $CP_F<RSW_s$ refers to a comprehensive program approach implemented in FORTRAN having a lower (better) ranking than redeveloped software in a special-purpose language. When no subscripts are present, the relation holds across any language choice - e.g., SB<CP means that the software builder is better than the comprehensive program regardless of language used.

Each of the seven criteria will now be considered. The ranking relationships will be stated and the reasoning behind that relation will be expressed.

3.2.1 REQUIREMENT FOR GLOBAL KNOWLEDGE OF APPLICATION SOFTWARE

$CP<SB<RSW$

The user of a comprehensive program could respond to prompts without detailed knowledge of how the model was implemented. A software builder would require more user knowledge to fit the parts of the model together. To use redeveloped existing software would demand the most knowledge of the model because the user would not be given any assistance in putting together existing programs.

$CP_F=CP_O=CP_S$

Because the user inputs are so restricted, the language used in the single program would not affect the rating significantly.

$RSW_S<RSW_O<RSW_F$

$SB_S<SB_O<SB_F$

It seems reasonable to assume that any special purpose language would be designed with this knowledge criterion in mind, so it would be better than other existing languages, many of which would (in turn) conceal model details better than FORTRAN.

3.2.2 REQUIREMENT FOR NEW SYSTEM KNOWLEDGE

$RSW_F < CP$

Users know FORTRAN already, so that redevelopment in FORTRAN would not entail learning any additional command language or support software. The next best choice would be a comprehensive program, which would handle all support and system commands while leaving the user only to learn sets of responses.

$SP_F = SP_O = SP_S$ Same as under "Knowledge" criterion

$SP < RSW_S < RSW_O$

The restricted range of responses in a comprehensive program would shield the user from learning system support commands. Redeveloping software in a language other than FORTRAN would require the user to interface with the software support environment in somewhat different ways than at present (e.g., different compilers, different conventions for accessing system libraries, etc.).

$RSW_O < SB$

The software builder would require the user to learn new commands to manage the support environment. This would be more difficult than the learning required in the case of redeveloped software because, even with a non-FORTRAN language, the commands would have something in common with the present user interaction-running compilers, using editors, etc.

$SB_F < SB_S < SB_O$

The FORTRAN familiarity would help the user the most. Next, a special-purpose language would be designed with this learning criterion in mind so it would presumably be easier than a non-FORTRAN existing language.

3.2.3 FLEXIBILITY AND ACCESSIBILITY

$RSW_F=RSW_O=SB_F=SB_O$

All four options provide the user with a procedural language which allows access to any details of the model.

$RSW_F=RSW_O=SB_O<RSW_S=SB_S$

The special-purpose languages may not permit access to some detailed aspects of the model because they would likely be designed to conceal some of the details from users.

$RSW_S=SB_S<CP$

A user will not have the flexibility of working in a language when interacting with a single program.

3.2.4 EASE OF APPLICATION LEVEL MODIFICATIONS

$P<SB<RSW$

The "prompt and respond" interaction with a comprehensive program would enable the user to make small changes very easily. The generalized input/output and separate parts in a software builder would require more effort but still less than the use of redeveloped existing software in which the user would need to work through language rules to make changes.

$CP_O=CP_F=CP_S$

Same as under "Knowledge" criterion

$SB_S<SB_O<SB_F$
$RWS_S<RSW_O<RSW_F$

It seems reasonable to assume that any special-purpose language would be designed so that small changes would be easier to make than in existing languages. FORTRAN lacks many features of other existing languages. Some of the features influence the effort to make changes, e.g., dynamic memory allocation, more data types, scoping, concurrency, etc.

3.2.5 EASE OF SYSTEM-LEVEL MODIFICATIONS

SB<RSW<CP

The organization of the software builder would make it easy to add new modules and integrate them into larger models. The simple interfacing in the software builder is missing from redeveloped existing software, making additions more difficult. A comprehensive program is intended to be all-encompassing, so it would not offer features for adding units to the model.

$SB_S<SB_O<SB_F$
$RSW_S<RSW_O<RSW_F$
$CP_S<CP_O<CP_F$

A special-purpose language would have some facility for adding program units because that feature is of interest to the users. It would be easier to add modules with several existing languages than with FORTRAN, because these languages offer a richer collection of subprogram types and supporting features.

3.2.6 DATA/SOFTWARE/ANALYSIS INTEGRATION

CP<SB<RSW

A comprehensive program would necessarily be a completely integrated package as part of its implementation. The software builder would have facilities for integrating support functions so it would be much better than existing software.

$CP_F=CP_O=CP_S$
$SB_F=SB_O=SB_S$
$RSW_F=RSW_O=RSW_S$

The particular language used would not matter significantly in determing how well the system was integrated with its support environment.

3.2.7 FEASIBILITY AND COST

$RSW_F < CP_F < SB_F$

$RSW_O < CP_O < SB_O$

$RSW_S < CP_S < SB_S$

The alternatives of redeveloping existing software require the least innovation to implement. Organizing the system as a comprehensive program would involve more thinking to consider approaches for integrating user flexibility and model completeness. A software builder is the most demanding to implement because of the introduction of generalized input/output, software pacts manipulation, and model management.

$SB_F < RSW_O$

This assessment hinges on the trade-off between effort to implement the features in the software builder while remaining with FORTRAN and the effort to redevelop existing software with no significantly new features but switching to another language for the implementation. The decision was made that the effort to switch languages would be greater than the effort to implement new features.

$SB_O < RSW_S$

The effort to implement new features in the software builder and use a non-FORTRAN language is substantial. However, the assessment was that this effort would be less than the effort involved in designing a special-purpose language and redeveloping existing software in that new language.

3.3 COMPARING THE ALTERNATIVES

From the rankings in Table 3-1, the software builder alternative used with the FDAS prototype is not clearly superior. The comprehensive program is best, followed by the software builder, with redeveloped software last. Regarding the choice of language within each approach, FORTRAN is ranked

best, followed by "other existing language" and special-purpose language.

It is unwise to rely exclusively on the numerical totals. Adding the rankings to produce a total for each alternative means that all criteria have equal weights, which is not generally true. For example, if "flexibility" and "ease of extending the system" are more heavily weighted then the comprehensive program would be penalized and the software builder made more attractive.

Some basic features of the assessment procedure must be recalled whenever results are discussed. For example, agreeing to use those particular seven criteria was a fundamental decision which obviously effects the assessment. Also, the ordinal measures correspond to equal intervals between successive ranks. Perhaps in a lengthier assessment period with more discussion, the assessment team might have been willing to commit to some rough interval measurement - e.g., how much better is the first choice than the second choice?

A "learning curve" characteristic of the criteria is less obvious but no less important to a fair reading of the results. Some criteria are significantly only with the first uses of the system while other criteria are uniformly applicable over the system's lifetime. For example, "minimize new system knowledge" is an issue when users begin to interact with the system. As the user becomes more familiar with the system, the importance of this criterion diminishes. "Feasibility and cost" is another criterion with an initial impact but no continuing relevance. In contrast, "flexibility" is assessed by considering features which are representative of the interaction with the system at any time during its life. If the two "short-term" criteria are deemphasized, the software builder alternative is more attractive.

Another consideration which the assessment team was unable to judge concerned the degree to which the alternatives are consistent with broader objectives of the using organization. In short, are the side effects beneficial or not? Consider an approach which calls for development in another existing language, say, Pascal. A lasting benefit to the organization would be the increased staff skills in Pascal programming, assuming the staff was not fluent in Pascal. Such development would also likely create a continuing need for Pascal skills for maintenance and enhancement. The same observation is true of other alternatives which involve a degree of innovation to develop the system. Experience implementing the features of the software builder is a benefit when similar features need to be introduced in other software. If a low ranking on "ease of implementing" is due to the innovative nature of the task, there may be a cause for reassessment to account for the benefit of increasing the skills of the staff who developed the innovative techniques.

In summary, the results clearly show that other alternatives are viable. The comprehensive program approach is the best one based on the rating procedure used, but a final determination depends on how the criteria are weighted and how important are other considerations discussed here, e.g., long-term benefits.

SECTION 4 - CONCEPTS AND IMPLEMENTATION OF FDAS

The FDAS prototype incorporates some important software engineering concepts. The purpose of this section is to review these concepts as they likely would be realized in the full implementation of the system. FDAS provides an integrated program development environment for non-expert programmers working on flight dynamics problems. In the terminology of Section 3, it is a "software builder". The software engineering concepts included in FDAS were selected to facilitate its function of enabling users to construct and execute programs with only a minimal knowledge of the encompassing systems.

The basic organizational concept of FDAS is the "experiment". It consists of three steps: assembling data and software in compatible formats; executing the software with the data; and analyzing the results. The experiment concept accurately reflects the viewpoint of the user. However, some consideration should be given to explicitly distinquish between data and software preparation in FDAS.

FDAS has three basic components that combine to produce an experiment (figure 1). Experiment data and experiment software are stored in libraries under the control of the experiment management system. An analyst uses this system to construct and execute an experiment. The results analysis step is not studied in this evaluation because it is not well defined in the FDAS prototype. The user interacts with the experiment management system via a series of menus and/or commands. The experiment software is composed in an extended FORTRAN language. These two features, the command language and extended FORTRAN are discussed in more detail in the following sections.

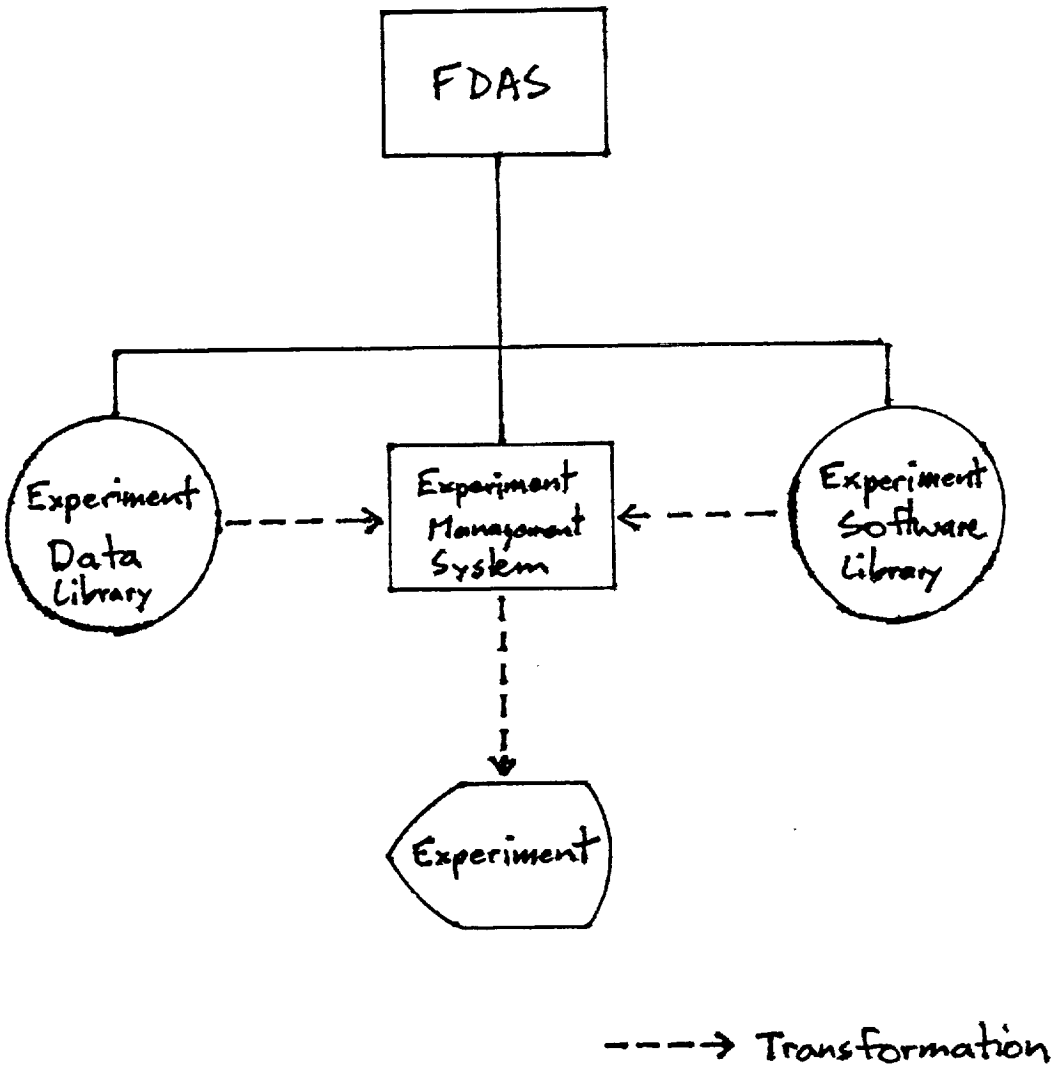

Figure 1. FDAS Organization

4.1 FDAS COMMAND LANGUAGE

The FDAS command language enables the user to retrieve software segments, modify software segments and data, and control the sequence of execution. An hierarchical menu system generates commands for the novice user. An awkward feature of the menu structure is the lack of a sequential path through it. That is, the user can only get to the next step (e.g., from preparation to execution) by first backing up. This is inconsistent with the basically sequential concept of "experiment". Consequently, it interferes with the novice's learning.

The FDAS command language could be an effective tool for the experienced used. The ability to save sets of commands as experiment control programs (ECPs) enhances the ease of use of the system. However, a false sense of continuity can be engendered by the use of ECPs. Two executions of the same ECP do not necessarily perform the same experiment. Changes to software and data are not always reflected in an ECP. In order to guarantee that an experiment is repeatable, the facility to save load modules must be available. Furthermore, the mixture of data and commands in an ECP can be confusing to the user.

Some of the terms used in the FDAS command language are nonstandard. The terms MODULE and COMPONENT, for example, are confusing to users. FDAS cannot be used effectively by someone with no programming knowledge. Anyone sophisticated enough to change a module or component will be expecting terms such as system, subsystem, and subroutine. Another term, ALIAS, is a misnomer. "Equivalence" would be more appropriate.

The MENU and HELP facilities, by themselves, are inadequate documentation. Relying on them is analogous to traveling across country aided only by a set of one-square-mile (each)

maps. It is difficult to determine where you've been or where your're going, even if the gas station you are parked at appears on one of the maps.

4.2 FDAS EXTENDED FORTRAN

The experiment software managed by FDAS is implemented in an extended version of the FORTRAN language (FPL). The special language features are provided by a preprocessor, FPL, which extends FORTRAN to include global system parameters, abstract data types, abstract data functions, and generalized input/output. These features facilitate software packaging and intermodule communications.

However, the abstract data types are limited to a subset of FORTRAN types (e.g., vectors and matrices are special cases of arrays). Although a data type such as the quaternion could easily be added to FPL, other basic types such as the set and stack are intractable. Also, the concept of abstract data types profitably could be extended to a higher level (e.g., experiment).

The generalized input/output feature provides a flexible method of intermodule communications. However, it does not provide any protection against unintentionally accessing and/or altering unrelated data. Instead of developing a preprocessor, this capability could have been implemented as calls to a subroutine package.

The system parameter, abstract data, and generalized input/output features could be provided in a less awkward and more powerful format (with less programming effort) by employing a language that includes abstract data types and scope definitions. PASCAL, for example, has a much wider range of data types than those provided by FPL. The scope feature of PASCAL allows a variable declared in the "root" of a program to be accessed without declaration anywhere else in the program.

SECTION 5 - CONCLUSIONS AND RECOMMENDATIONS

The major conclusions and recommendations of this evaluation group, presented in detail below, can be summarized as follows:

The FDAS concept has substantial merit and should be pursued.

The specific approach of the current FDAS development (which we have termed a "software builder" alternative) is a feasible concept, but not clearly superior to other approaches.

The FDAS prototype is unsatisfactory as a prototype; it carries untested design assumptions to excessive detail, so that evaluations are performed on the detail - not on the assumptions.

The FDAS prototype is unsatisfactory as a minimal set of user interaction capabilities; it is difficult to use, its behavior is unpredictable, its terminology confusing. The problems are not with the lack of those functions that are not available in the prototype, but with the intractability of those that are.

We recommend that the requirements definition effort be pursued, and that several alternative approaches be studied (chief among them: a software builder approach using Pascal; and a comprehensive data-driven program approach).

Consideration should be given to a new prototyping effort with goals and criteria much more carefully specified - the prototype should not be a "limited capability" version of the planned full system, but a mock-up for testing user-interaction concepts.

5.1 DESIRABILITY OF THE BASIC FDAS CONCEPT

The various requirements documents and data make a strong case for the replacement of existing clumsy, ill-suited software with a coordinated support environment. Current research and practice, in areas as varied as DoD's planned Ada (tm) Programming Support Environment (APSE) and integrated microcomputer software (e.g., Lotus 1-2-3) indicate the need and user acceptability of such functional coordination. The areas addressed by FDAS have so much in common that a unified analytical tool has a substantial amount of leverage in terms of reused software. Comparable (though not equivalent or transportable) support tools are effective in other areas; flight dynamics analysis activities can almost certainly benefit as well.

This evaluation team approached this assessment with a moderate to strong positive bias toward the concept and goals of FDAS; our review and research have, if anything, strengthened this bias. We strongly recommend pressing forward with a requirements definition effort for FDAS.

5.2 THE SOFTWARE BUILDER ALTERNATIVE

This group, working from the initial problem statement and searching literature and experience for examples, identified several feasible approaches to designing an FDAS full system. We developed a rating scheme to compare these approaches across initial requirements. We found that the "software builder" alternative embodied by the current software effort is only one of several possibilities. We concluded that many desirable features could better be provided with either a different base language (e.g., Pascal) which would facilitate provision of data abstraction, global common data, and modular program element construction capabilities; or with a comprehensive data-driven program which might obviate altogether the need for analysts to be programmers. The effort

required to convert existing software to the FDAS-prototype extended FORTRAN approach was projected to be so large that the penalty for converting to an entirely different language or mechanism would be small relative to the total effort.

We recommend that a requirements definition effort be directed toward defining requirements in terms applicable to several approaches, and that several approaches be seriously considered and assessed. We recommend further that particular attention be paid to the language issue - the advantages of "coming" languages (Pascal or Ada), in terms of modularity, transportability, and availability of trained programmers in coming years are substantial.

We also recommend that more detailed profiles of the intended user and uses of the FDAS be developed. If the requirement that the user need not be a software expert is taken seriously, the value of the "software builder" approach is significantly reduced.

5.3 THE FDAS PROTOTYPE AS REQUIREMENTS DEFINITION TOOL

In software design a prototype should serve two purposes: it should aid in testing the utility of proposed capabilities, and support the "proof of concept" of those capabilities with demonstrable value (Refernces 9 and 10). The FDAS prototype seems to have addressed the second problem without adequately treating the first. Instead of a testbed where concepts and terminology (such as "experiment" and "module" and "alias") can be tested at small cost, the Release 2 system provides a demonstration that these elements can be implemented (albeit with greater effort than initially projected). The level of information to be derived from such a detailed prototype is further into design than is warranted by the state of requirements defintion. The prototype can indicate that this particular menu system is not user-friendly; but it does not provide guidance on how better to build one.

The goals of the prototype (the questions it was to help answer) were not formulated in a manner conducive to objective results. The apparent guideline was that the prototype should provide a subset of the full FDAS functionality; the "prototype" characteristic is reflected in that no "hooks" were included for later development and that software instrumentation was incorporated. A better definition of "prototyping" should have been employed.

This group recommends that a somewhat different orientation be used on future prototyping efforts, and that this project not be used as a model for such efforts.

5.4 THE RELEASE 2 SYSTEM AS IMPLEMENTATION

It can be difficult, when assessing a prototype, to distinquish between design characteristics and artifacts of implementation. We have tried to focus on essential elements of the FDAS approach selected to date, but cannot warrant that we have succeeded in all cases. We identified as central to the selected design the user interface approach, the data/software management capability (the executive), and the language extension provided by the FPL preprocessor.

The menu and command-driven user interface does not meet its goals of ease of use and user friendliness. The automatic sequence is not clear or predictable, the contextual clues vague or non-existent, the terminology confusing. No user should be propelled into the system editor (with an active file) without specifically requesting it; defaults should not lead in circles. We intended to build a small FORTRAN system as a test of the data management and analysis capabilities; we were unable adequately to master the use interface.

The terminology selected for the data management executive (chosen somewhat to be distinct from software development

terms) is unclear. It was apparent that the data management function was complex, and it appeared to work; but the user interface was an obstacle.

The capabilities procided by the FPL preprocessor (notably data abstraction and standardized I/O) are highly valued in the software engineering world (though no objective studies have been reported), but could have been provided with other means. The justification for the preprocessor is unclear.

We recommend that the user interface be completely respecified and redesigned; that the data management structure be more rationally organized; and that the entire question of the need for a preprocessor be reevaluated.

REFERENCES

1. J. E. Sammet, "Roster of Programming Languages," ACM SIGPLAN Notices, September 1972; pp. 3-12.

2. C. R. Hollander and Y. Iwasaki, "The Drilling Advisor," Proceedings of the IEEE Spring COMPCON. New York: Computer Societies Press, 1983; pp. 116-119.

3. L. Forman, "The New York Times Corporate Planning Model," Proceedings of the Winter Simulation Conference. New York: Association for Computing Machinery, 1976, pp. 437-448.

4. M. D. Mesarovic, et. al., Mankind at the Turning Point. New York: E. P. Dutton, 1974.

5. B. Meyer, "Principles of Package Design," Communications of the ACM, volume 25, number 7, July 1982, pp. 419-424.

6. J. F. Isner, "A Fortran Programming Methodology Based on Data Abstraction," Communications of the ACM, volume 25, number 10, October 1982, pp. 686-697.

7. M. M. Zloof and S. P. deJong, "The System for Business Automation (SBA): Programming Language," Communications of the ACM, volume 20, number 6, June 1977, pp. 385-395.

8. P. Bassett and J. Giblon, "Computer Aided Programming (Part I)," Proceedings of the IEEE SoftFair. New York: Computer Societies Press, 1983, pp. 9-20.

9. R. E. Mason and T. T. Carey," Prototyping Interactive Information Systems," ACM Communications, volume 26, number 5, May 1983, pp. 347-354.

10. L. Sharer, "The Prototyping Alternative" Programming, 1983.